CS 677: Big Data

Network Design

Lecture 8

Today's Schedule

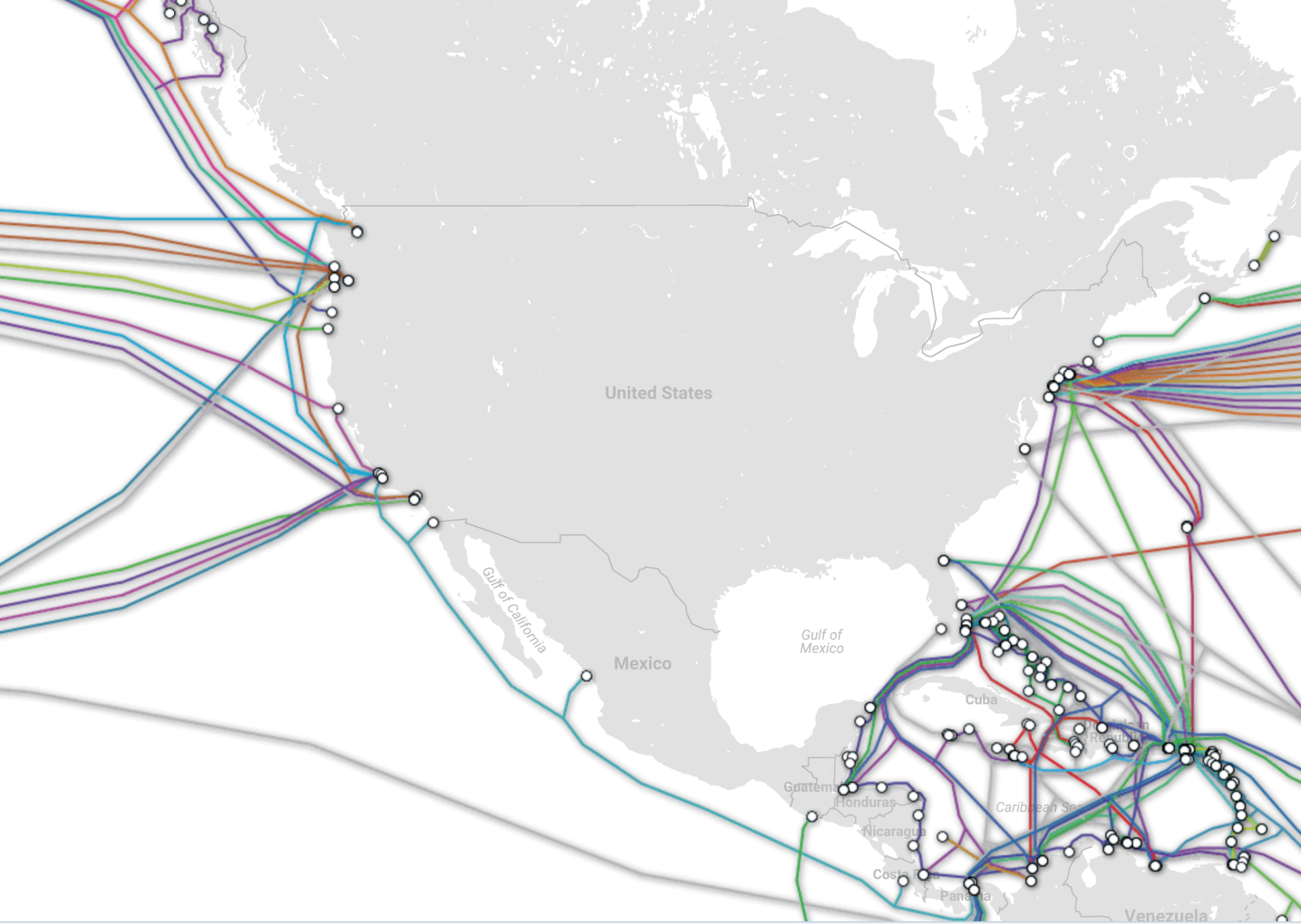
- Network Designs
- Scaling our Networking Code

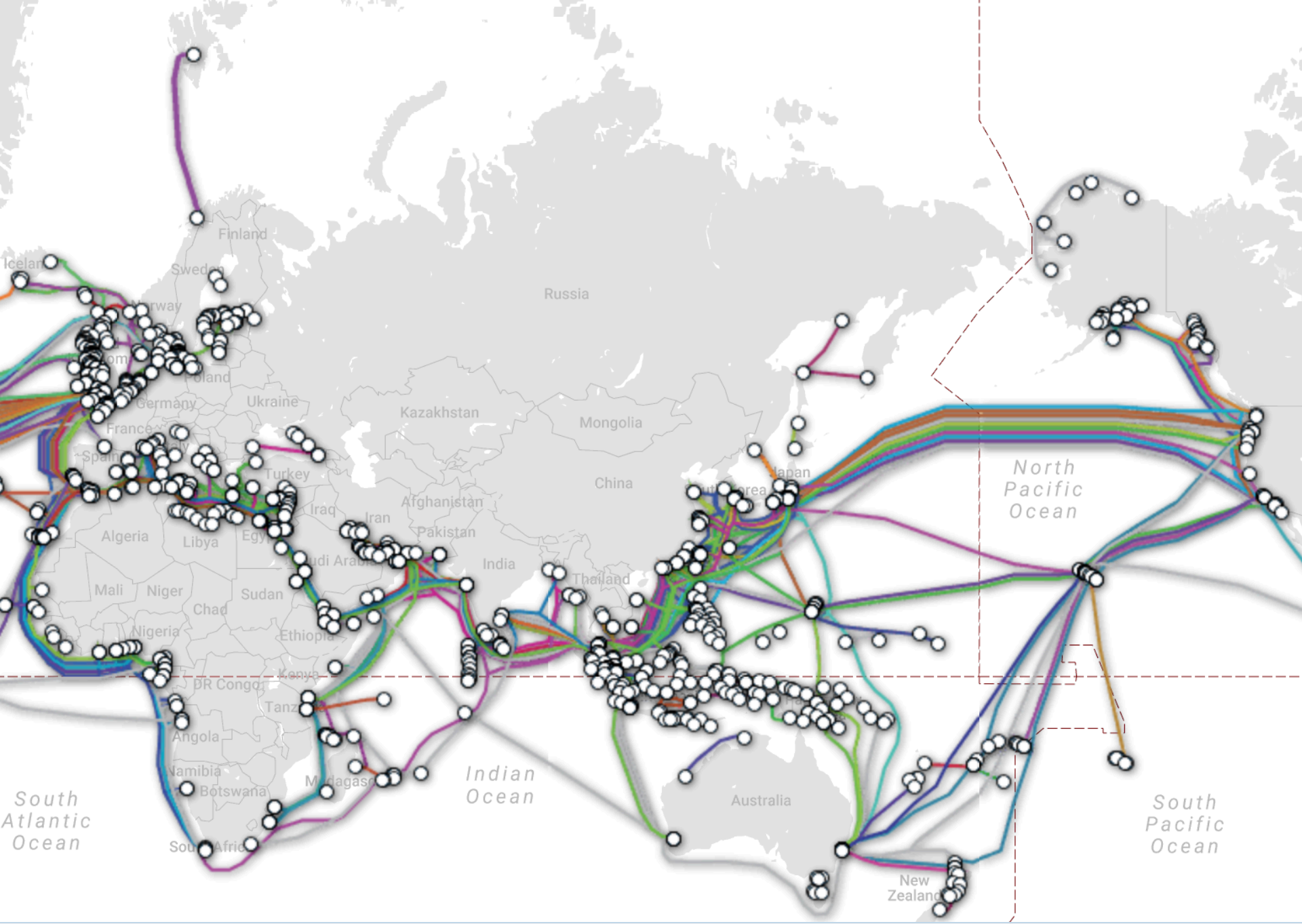
Today's Schedule

- **Network Designs**
- Scaling our Networking Code

Thinking About the Network

- For the most part, we can rely on the network to do its job and we'll live at a higher level of abstraction
- Many networking concerns still creep up when designing our systems
  - For instance, do we use TCP or UDP?
- We still need to think about **bandwidth**, **latency**
- …especially when dealing with multiple data centers
  - https://www.submarinecablemap.com

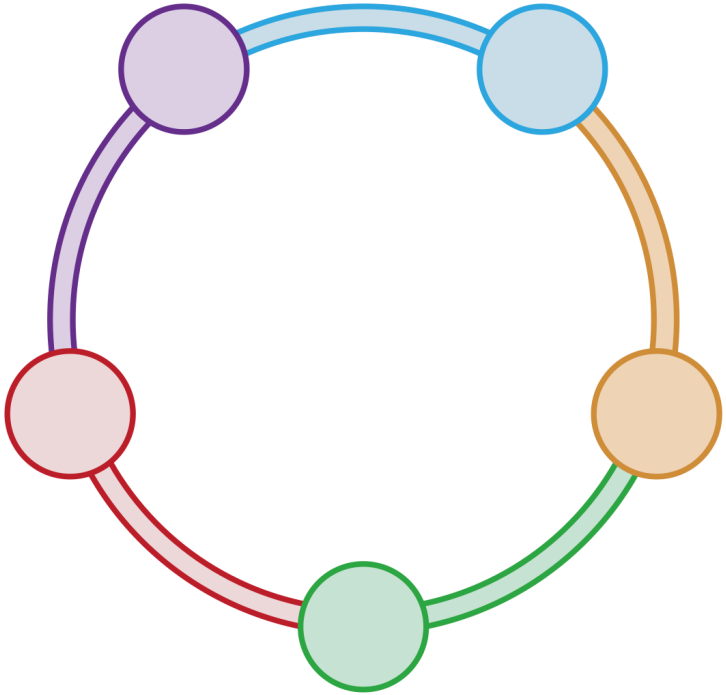

United States
Mexico
Gulf of California
Gulf of Mexico
Cuba
Honduras
Nicaragua
Venezuela


Finland
Russia
Germany
Ukraine
Kazakhstan
Mongolia
France
Turkey
China
Iraq
Iran
Afghanistan
Pakistan
Algeria
Libya
India
Thailand
Mali
Niger
Sudan
Chad
Nigeria
Ethiopia
Kenya
Angola
Namibia
Botswana
Australia
New Zealand
North Pacific Ocean
South Atlantic Ocean
Indian Ocean
South Pacific Ocean

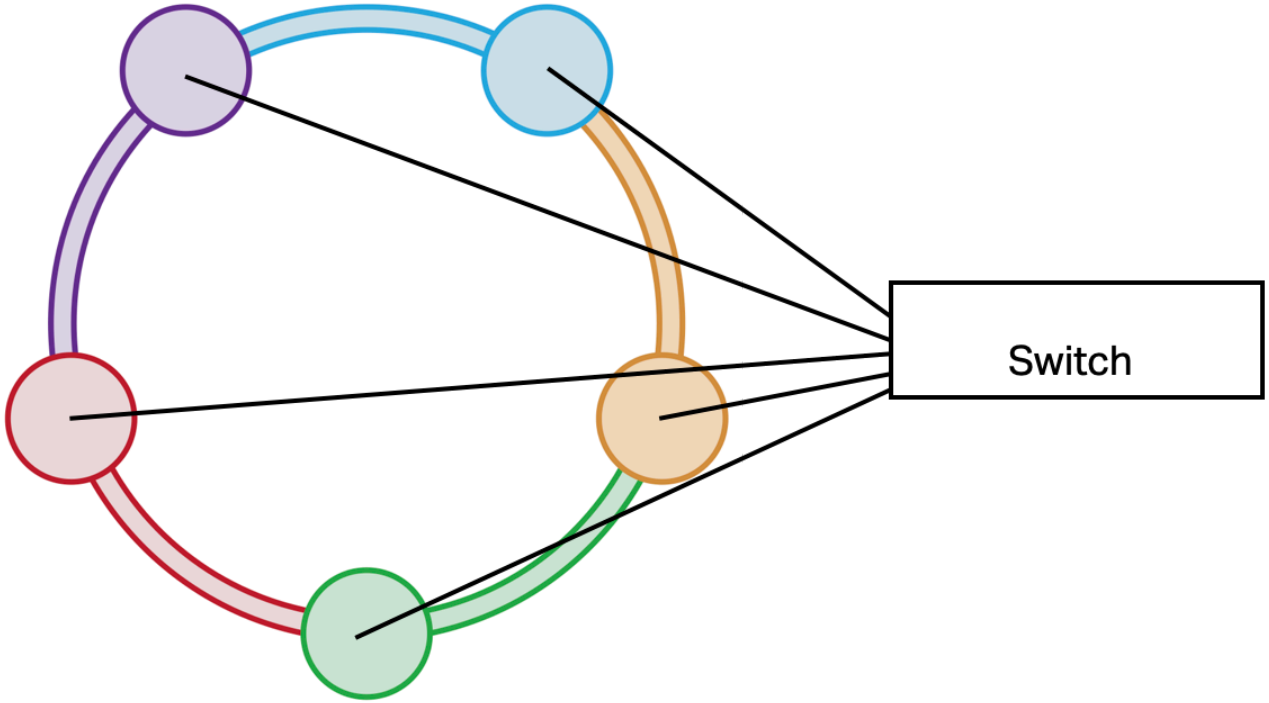
Designing the Network

- The messages exchanged between nodes are influenced by the underlying network design
  - A P2P network operates differently than set of **NFS** servers
  - Join notifications, file locking, periodic heartbeats
- Note that this generally doesn't mean the *physical* network
- Instead, we often design distributed systems around **overlay** networks

Overlay Network: Ring Topology



Ring Topology: Physical Network



Ring Messages

- In a ring topology, we can get by with one message:
  - Send to next node
- …Or, if you want to get fancy:
  - Send to next node
  - Send to previous node
- This setup only makes sense in certain situations
- It certainly isn't optimal for searching for a specific file/user/etc!
  - What might a good use case be?

Shared Nothing Architecture

- A popular design for distributed systems is the **shared nothing** (SN) architecture
- Each node in the system is self-sufficient
  - No specialization
  - **No centralized components**
- SN helps ensure scalability
- When all the nodes are the same, failure cases are easier to handle

Microservice Architecture

- Shared nothing design inspired loosely-coupled distributed architectures based on **microservices**
  - A server process that does one thing and does it well
- Advanced data workflows can be built by creating pipelines of microservices
- This approach was championed heavily at Amazon and likely led to their runaway success with AWS

Bezos' Mandate [1/3]

- There was an internal post by a former Amazon employee (now at Google) that accidentally went public
- In 6 years, Amazon went from an online bookstore to a $1bn IaaS provider
  - Google should've owned the cloud market! How did this happen? Why is Google unable to build quickly?
- Bezos *reportedly* issued a mandate to his company in the early years that built a foundation for services success
  - (when he wasn't wearing cowboy hats and blasting off into space…)

Bezos' Mandate [2/3]

- All teams will henceforth expose their data and functionality through service interfaces.
- Teams must communicate with each other through these interfaces.
- There will be no other form of inter-process communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.

Bezos' Mandate [3/3]

- It doesn't matter what technology you use.
- All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.
- Anyone who doesn't do this will be fired. Thank you; have a nice day!
  - (This part is *probably* made up… 😄)

Business Advantage

- This approach makes it obvious why Amazon can build new services so quickly
- Things get built internally and already have a “public”-ready interface
  - Get tested internally
- Want to sell it as a service? Make a UI, spin up instances to support the service, and sell it

The Downsides of Shared Nothing

- Complicated development: assuming all nodes are the same means dealing with many corner cases
  - What if the underlying hardware is different?
  - Sometimes it's just easier to put some information in a central repository
- If excessive state information must be transferred, the system will be more susceptible to latency
  - Same goes for algorithms that require more coordination
- There is a very recent trend back towards 'monoliths'…

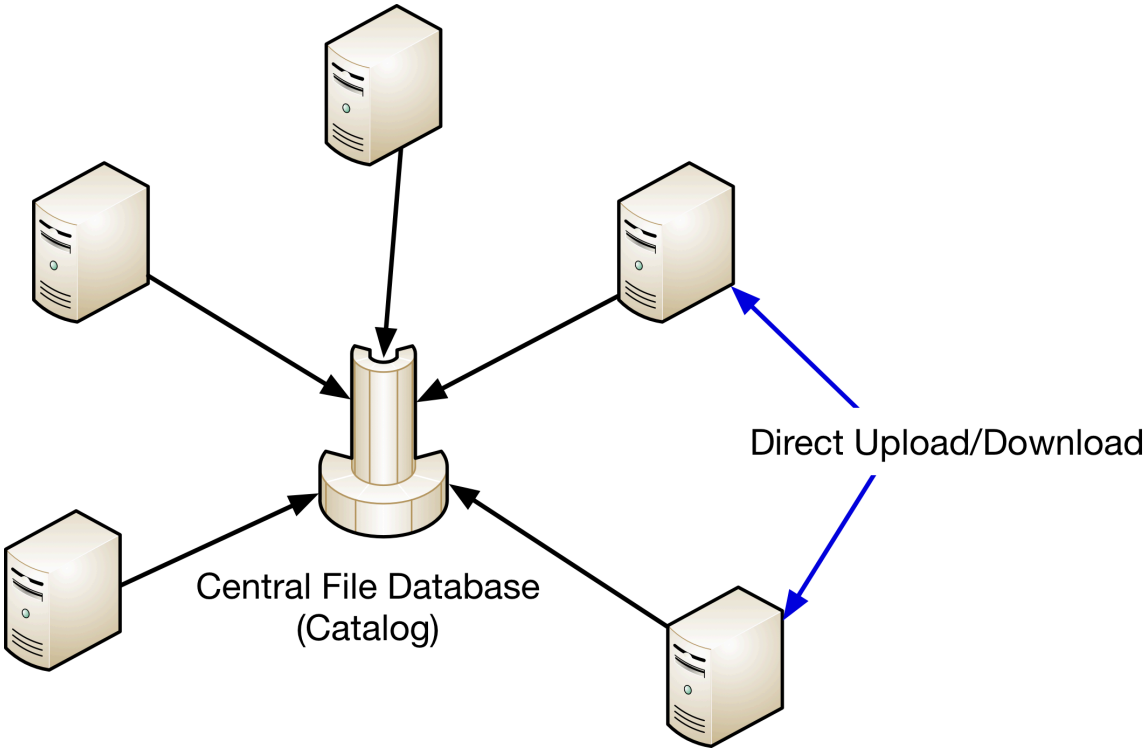
(Semi)-Centralized Networks

- An alternative is to include centralized/specialized components
  - Far simpler
  - Represents a single point of failure
- GFS: central catalog of all files
- To help deal with failures, hierarchical designs spread out the state information
  - Example: DNS

Napster (P2P)

- The original Napster (not the streaming service) was one of the first popular P2P file sharing tools
  - Laid the groundwork for today's distributed systems
- On startup, the Napster client transmitted its list of files to a central database
  - Napster didn't host any files, it just maintained a database
  - The paradigm was a bit revolutionary at the time

Napster Network Layout



Napster's Database

- The central database ended up being Napster's undoing
- As the service got popular, the server was flooded with requests and couldn't keep up
- It also was an easy target for law enforcement to shut down

Gnutella

- Learning from Napster's mistakes, *Gnutella* was designed to be completely decentralized
- Instead of using a central database, queries **flooded** through the entire network
  - Unstructured network
- This poses some problems though:
  - Performance: querying gets slower with more users
  - How do you know when your query is over?

KaZaA

- Gnutella didn't catch on, but *KaZaA* did
- Struck a balance between the two approaches
- **Supernodes** functioned as local databases
- Client queries were sent to the supernodes, which would also query their peers
- In other words: a hierarchical approach

Today: BitTorrent

- Nowadays most folks download their favorite non-copyrighted songs, public domain media, and Linux distributions using BitTorrent
  - But certainly not copyrighted material
- Puts the responsibility of indexing the files on websites/users
- Recent versions support **magnet**, backed by a distributed hash table
- IPFS employs a similar paradigm for sharing data

Today's Schedule

- Network Designs
- **Scaling our Networking Code**

Scaling Threads

- A common paradigm when designing servers is the **fork-join** model
- Split off when work can be handled by a single thread, then merge back together when shared state is necessary
- Web server: for each incoming connection, start up a new thread
  - The thread handles direct communication with the client

Thread Overhead

- It takes time to create threads
  - Updating OS data structures, creating a new lightweight process
  - **Overhead** from thread context: each has an individual stack
- Generally in this model when a thread is performing I/O, it **blocks**
  - Has to be context switched out so others can use the CPU for useful work

Apache / C10K Problem

- One of the most popular web servers, Apache, followed this model
  - Thread-per-connection
- The server would famously crumble when hit with huge amounts of traffic
  - Their main competitor at the time, Microsoft IIS, wasn't much better
- This led to the formulation of the C10K problem
  - "It's time for web servers to handle ten thousand clients simultaneously, don't you think? After all, the web is a big place now."

An Update: C10M

- On modern hardware, you can achieve 10,000 connections using Visual Basic 6 and a network interconnect composed of carrier pigeons
  - See RFC 1149, *IP over Avian Carriers*
- Around ~2010 WhatsApp demonstrated 2m active connections on a single 1U server
  - 24 cores running Erlang on FreeBSD

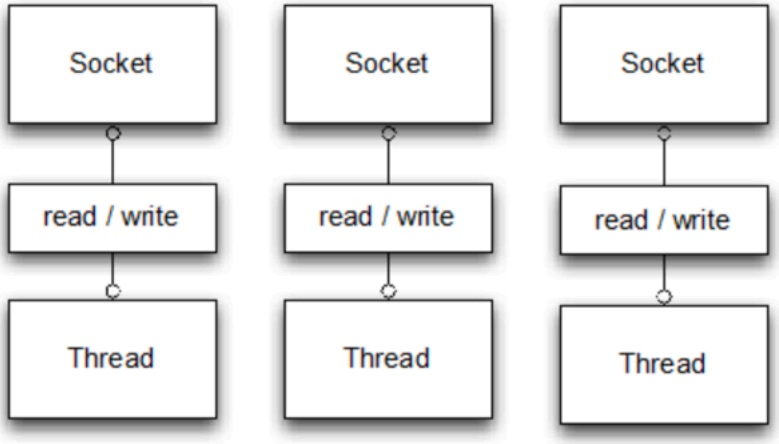
How? Asynchronous I/O

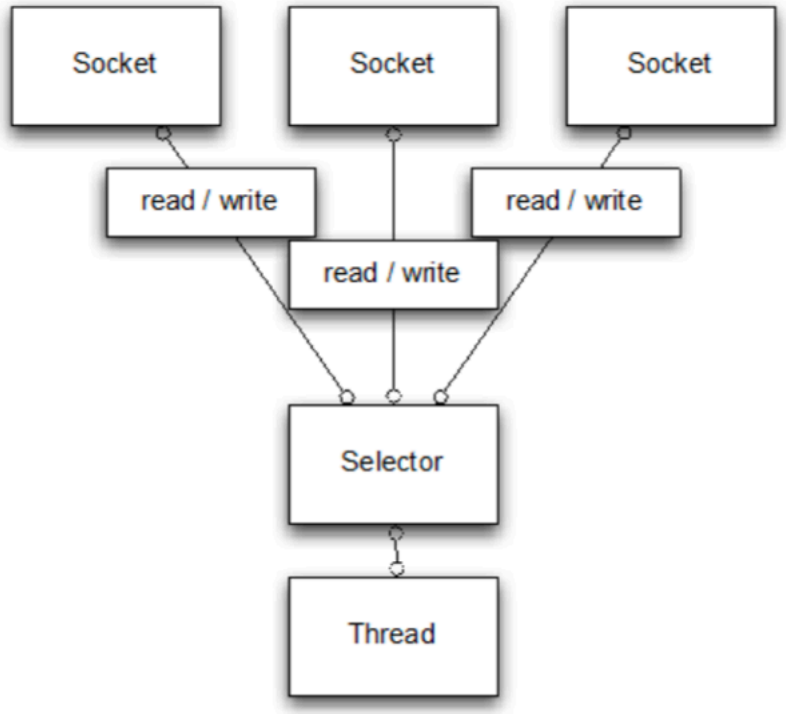
- Usual setup: we're sending a 5 MB file. Start writing it to a socket, which blocks our thread
  - OS context switches out the thread, something else runs on that CPU/core
- Rather than blocking, wouldn't it be better to tell the OS "send this message," and then go work on something else?
  - This is the idea behind asynchronous (or non-blocking) I/O

Async I/O in Java

- Java is the most widely-used language for building scalable distributed systems
  - **(FOR NOW! 😉)**
- There an `nio` package that provides non-blocking I/O functionality
  - Wrapper over OS mechanism (Unix select, Linux epoll, FreeBSD kqueue)
  - Notoriously hard to use
- Many folks use the **netty** library (built on top of nio) or something similar

Sync vs. Async



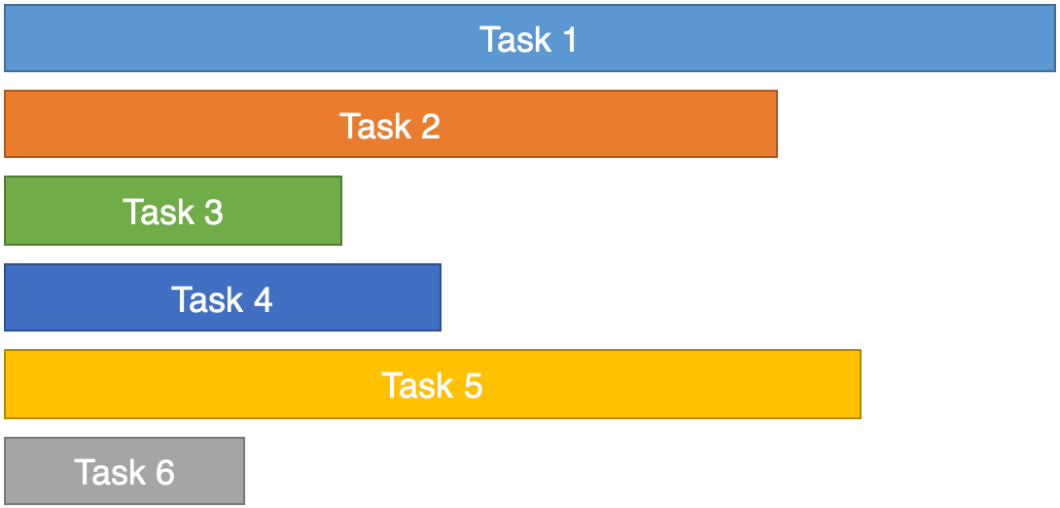


Maurer & Wolfthal. *Netty in Action*

How it Works: Server Side

- Connections are registered with the OS `select()` mechanism
- The server blocks on the selector rather than an individual socket connection
- When data is available, the selector gives the server a list of channels that are ready to read
  - Includes other states as well
- The server reads from the channels, buffering data if necessary
- Once a complete message has arrived, it gets passed to a thread pool for handling
- Fun fact: this is more or less the way node.js operates

Blocking vs. Non-Blocking



- Assume tasks arrive in numeric order all within 1 ns of each other.
- The width of the bar indicates how much data is being received
- Computational costs are proportional to task size
- Which task finishes first for blocking and non-blocking I/O?

Asynchronous Send

- Blocking: send(data):
  - wait until ‘data’ is sent
- Non-blocking: send(data):
  - returns immediately even though the send hasn’t finished yet
  - (or maybe it hasn’t even started)
- How do we know when the operation actually finished though?

Futures

- If we have to eventually know the result of an asynchronous task, we can use **futures**
- Say we send a message, start doing some other work, and then want to make sure the recipient got the message
  - Future.getResult()
    - This operation will block if the send hasn't finished yet

      We have more control over where we actually block, and if the send is already done it doesn't slow us down!

Operating System Mechanism

- At a 1000 ft view, this functionality requires the OS kernel to send an **event** to indicate a socket (or file descriptor) is ready
- Naturally leads to event-based systems
  - Each **action** is an event
  - Each runtime-level thread maps to an OS-level thread that blocks
- Note: this programming model can be cumbersome
  - Alternatives: green threads, fibers, actors

Actors and Green Threads

- The **actor** pattern can be used to encapsulate events and represent how they are processed between entities
- **Green threads** are runtime-level threads; they are not backed by an OS thread
- Oracle et al. are incorporating green threads into the JVM
  - Allows more control over scheduling, requires less heavyweight OS resources to run
- Many other languages are adding support for this (if they don't have it already)
  - Async/Await style programming? Avoid!
- You might be wondering what the concurrency story is in Go…

goroutines

- Rather than mapping each Go thread to an OS thread, the runtime:
  1. Allocates a flexible number of worker threads
  2. Schedules green threads (called *goroutines*) itself
     - goroutines **yield** when they are about to block, letting a different goroutine to take over the OS-level thread
- Effectively allows the programmer to write *blocking* code that operates in a *non-blocking* way behind the scenes!
  - It is fairly well-established that blocking code is much easier for programmers to understand and reason about

Launching a goroutine

- `go functionName()` will run `functionName()` as a goroutine, which essentially acts like a separate thread
  - Whatever comes after the `go` statement will run immediately as well, i.e., there is no blocking
- Like other languages, you *might* need to think about thread safety
  - But the recommended approach is to avoid situations that need thread safety
    - 🤔

Communicating Sequential Processes (CSP)

- Go's concurrency model is based on *Communicating Sequential Processes*
- The basic idea: goroutines share data by **communicating**, not via shared memory
  - Shared memory can be very fast, but is also the source of many concurrency problems
- To share data between goroutines, use a **channel**

Channels

Here's an example from Go by Example:

```
package main
import "fmt"

func main() {
    messages := make(chan string)
    go func() { messages <- "ping" }()
    msg := <-messages
    fmt.Println(msg)
}
```

What will this program output?

Idiomatic Go vs. Getting Done

- It’s great to write perfect, idiomatic Go code that uses channels for communication between goroutines
  - I would encourage you to research and follow best practices for Go – not just general best practices
- However, sometimes you’ll need things like mutexes to get your work done
  - That’s okay
- Don’t forget to balance learning the new language with GSD (getting stuff done!)